Improving the Programmability of STHORM-based Heterogeneous Systems with Offload-enabled OpenMP

Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, Luca Benini
University of Bologna
Viale Risorgimento 2
40136 Bologna - Italy
{a.marongiu, alessandro.capotondi, giuseppe.tagliavini, luca.benini}@unibo.it

ABSTRACT

Heterogeneous architectures based on one fast-clocked, moderately multicore "host" processor plus a many-core accelerator represent one promising way to satisfy the ever-increasing GOps/W requirements of embedded systems-on-chip. However, heterogeneous computing comes at the cost of increased programming complexity, requiring major rewrite of the applications with low-level programming style (e.g, OpenCL). In this paper we present a programming model, compiler and runtime system for a prototype board from STMicroelectronics featuring a ARM9 host and a STHORM many-core accelerator. The programming model is based on OpenMP, with additional directives to efficiently program the accelerator from a single host program. The proposed multi-ISA compilation toolchain hides all the process of outlining an accelerator program, compiling and loading it to the STHORM platform and implementing data sharing between the host and the accelerator. Our experimental results show that we achieve very close performance to hand-optimized OpenCL codes, at a significantly lower programming complexity.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design, Performance

Keywords

many-cores, heterogeneous SoCs, OpenMP

1. INTRODUCTION

The ever-increasing demand for computational power has recently led to radical evolutions of computer architectures to satisfy this need. Two design paradigms have proven particularly effective in increasing performance and energy efficiency of compute systems: architectural heterogeneity and many-core processors.

MES '13, June 23 - 24 2013, Tel-Aviv, Israel

A common embodiment of architectural heterogeneity is a template where a powerful general-purpose processor (usually called the *host*), featuring sophisticated cache hierarchy and full-fledged operating system, is coupled to programmable many-core accelerators composed of several tens of simple processors, where critical computation kernels of an application can be offloaded to improve overall performance/watt. Unfortunately, these advantages are traded-off for an increased programming complexity: extensive and time-consuming rewrite of applications is required, using specialized programming paradigms such as OpenCL [8].

OpenCL aims at providing a standardized way of programming such accelerators. However it offers a very low-level programming style. Programmers are responsible for writing and compiling separate programs for the *host* system and for the accelerator. Data transfers to/from the many-core and synchronization must also be manually orchestrated. In addition, OpenCL is not performance-portable. Specific optimizations have to be re-coded for the accelerator at hand to achieve performance.

This generates the need for a higher-level programming style [6] [18] [20]. Recently, the predominance of accelerator-based architectures has originated discussion within the OpenMP [14] architecture review board, which has included in the latest draft specification [22] a proposal for extensions to manage accelerators. Along the same line, Intel Xeon Phi coprocessors offer all standard programming models that are available for Intel Architecture, e.g. OpenMP, POSIX threads or MPI [16]. This is very convenient for programmers, to which the accelerator appears like a symmetric multiprocessor (SMP) on a single chip. The programming effort is significantly reduced, since no additional parallelization paradigm like OpenCL needs to be applied.

In the embedded domain such proposals are still lacking, but there is a clear trend towards designing embedded SoCs in pretty much the same way it is happening in the HPC domain [19] [20], and which will eventually call for the same programming solutions. The STMicroelectronics STHORM chip [12] is an example of a many-core accelerator meant for heterogeneous embedded designs. STHORM relies on a multi-cluster SoC, where each cluster consists of several tightly coupled cores (based on the STxP70 ISA) and L1 memory, with uniform memory access time. Inter-cluster communication travels through a NoC and is subject to non-uniform memory access (NUMA) effects.

In this paper we present a programming model, compiler and runtime system for a prototype board from STMicroelectronics featuring a ARM9 *host* system and a STHORM many-core accelerator. The programming model consists of an extended OpenMP, where additional features allow to efficiently program the accelerator from a single *host* program. This is achieved by simply enclosing code blocks to be offloaded within a custom directive. Within such directive any standard OpenMP feature can be used. This al-

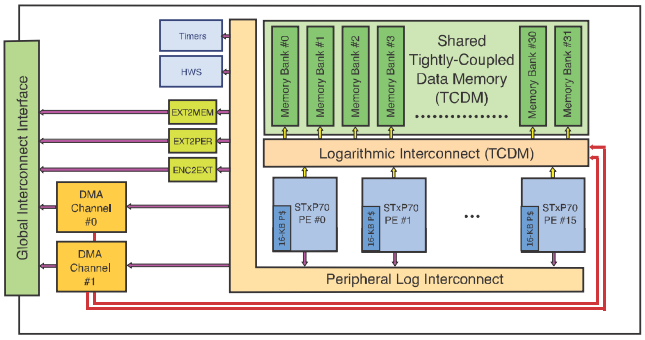

Figure 1: Diagram of a STHORM cluster

lows to conveniently program the accelerator itself through OpenMP parallelization, and to synctactically specify this "sub-OpenMP accelerator program" from within the main *host* application, rather than manually separate *host* and accelerator parts. The OpenMP directives allowed within an offload block provide additional extensions to distribute the workload among STHORM clusters in a NUMA-aware manner, thus improving the performance.

We developed a single, multi-ISA compilation toolchain based on the integration of GCC (which provides robust, open-source OpenMP implementation, and reliable ARM backend) and LLVM (with optimized STxP70 backend). This integrated toolchain hides all the process of outlining an accelerator program from the *host* application, compiling it for the STHORM platform, offloading the execution binary and implementing data sharing between the *host* and the accelerator.

Two separate OpenMP runtime systems are developed, one for the *host* (which simply extends the original GCC GOMP implementation [13] with additional functions for offload) and one for the STHORM accelerator. The latter relies on a full-custom library, which provides lightweight support for multi-level fork-join parallelism and NUMA-aware thread-to-core mapping.

Our experiments compare the performance of representative benchmarks parallelized with our OpenMP and with OpenCL, natively supported by the STHORM platform. Results confirm that our approach achieves very close performance to hand-optimized OpenCL codes, at a much lower programming complexity.

The rest of the paper is organized as follows. In Section 2 we describe the STHORM chip and the heterogeneous system-on-board. In Section 3 we describe our programming model, compilation toolchain and runtime environment. In Section 4 we provide experimental evaluation of the proposed OpenMP implementation. In Section 5 we discuss related work. Section 6 concludes the paper.

2. TARGET ARCHITECTURE

This section describes the STHORM-based heterogeneous system targeted in this work. We first describe the STHORM SoC architecture, then the board and the heterogeneous system as a whole.

2.1 The STHORM many-core architecture

STHORM, previously known as Platform 2012 [12], is a project led by STMicroelectronics to build an area- and power-efficient many-core computing fabric. STHORM is a globally asynchronous, locally synchronous (GALS) fabric of multicore *clusters*. A STHORM *cluster*, whose block diagram is shown in Figure 1, is locally synchronous and contains a multi-core computing engine called *ENCore*, a *Cluster Controller* processor (based on the ST xp70 ISA), a

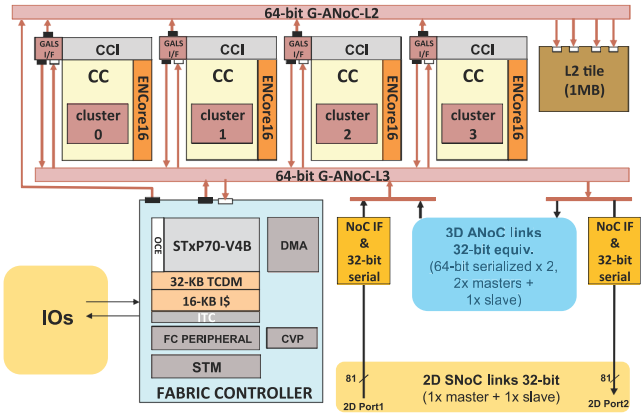

Figure 2: 4-cluster STHORM fabric

Clock Variability and Power (CVP) module and 2 DMAs.

Each *ENCore* computing engine contains (up to) 16 STxP70-v4 *Processing Elements* (PEs), each of which has a 32-bit RISC load-store architecture, with dual issue and a 7-stage pipeline. The PEs have independent instruction issue, therefore the *ENCore* can execute threads in a MIMD fashion. Moreover, each PE features a private instruction cache (16KB). All the PEs within a *ENCore* system communicate through a shared multi-ported, multi-bank tightly-coupled data memory (TCDM). This L1 memory has a size of 256 KB, and is explicitly managed by the software (i.e., it is a scratchpad memory, not a HW-managed data cache). The interconnection between the PEs and the TCDM was explicitly designed to be ultra-low latency [15]. It supports up to 16 concurrent processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). The ENCore also provides a hardware synchronizer and an interrupt generator.

Figure 2 depicts the whole many-core fabric. The STHORM fabric is composed of four *clusters*, plus a *fabric controller* (FC) tile, responsible for global coordination of the clusters. The FC tile consists of an xP70 processor, with local instruction cache and data TCDM, CVP module, DMA engine and several I/O interfaces. The FC and the clusters are interconnected among them and to L2/L3 memory via two asynchronous networks-on-chip (ANoC). The first ANoC is used for accessing a multi-banked, multi-ported L2 memory. The second ANoC is used for inter-cluster communication via L1 TCDMs and to access the off-chip main memory (L3 DRAM). Note that all the TCDMs, the L2 and L3 memories are mapped into a global address space, so every PE in the many-core has direct access to every memory device. Transactions to the main (L3) memory are transported off-chip via a synchronous NoC link (SNoC).

2.2 The heterogeneous SoC

The first embodiment of a STHORM-based heterogeneous system is the prototype board shown in Figure 3. This system is based on the Xilinx Zynq 7000 FPGA device, and features a ARM CA9 dual core *host* processor, main (L3) DDR3 memory, plus programmable logic (FPGA).

The ARM subsystem on the ZYNQ is connected to a AMBA AXI interconnection matrix, through which it accesses the DRAM controller. The latter is connected to the on-board DDR3 L3 memory via fast communication interfaces. As described in the previous subsection, the STHORM SoC is interfaced to the outer world through master and slave SNoC ports. To allow transactions generated inside the STHORM system to reach the L3 memory, and transactions generated inside the ARM system to reach internal STHORM L1 and L2 memories, part of the FPGA

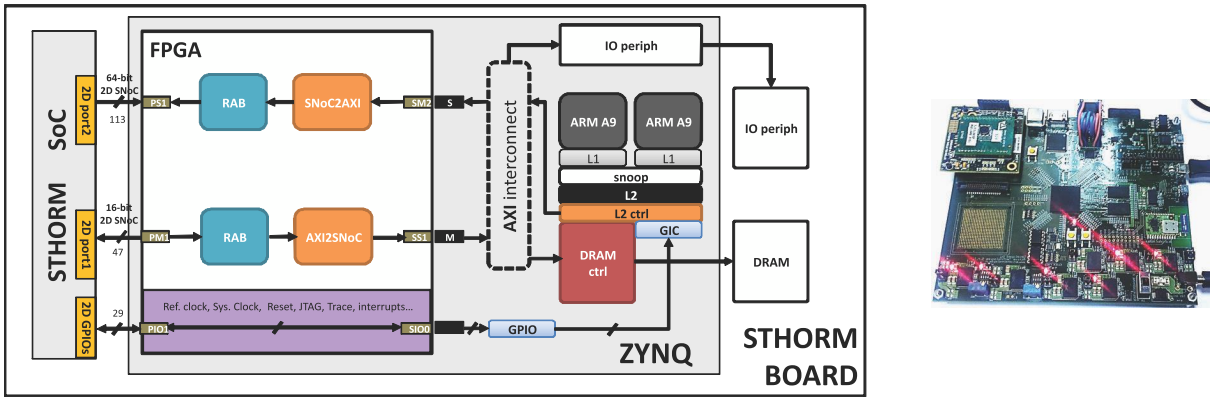

Figure 3: STHORM-based heterogeneous system.

area is used to implement a bridge to/from the STHORM chip. This bridge implements three main functionalities. First, it translates outbound STHORM transactions from the SNoC protocol into the AXI protocol, and injects them into the AXI interconnect matrix. Similarly, outbound ARM transactions are translated from the AXI protocol into the SNoC protocol. Second, it implements address translation logic in the remap address block (RAB). This is required to translate addresses generated from STHORM into virtual addresses as seen by the *host* application and vice-versa. Indeed, the *host* system features paged virtual memory and MMU support, while STHORM operates on physical addresses. Thus, the RAB acts as a very simple IOMMU. Third, it implements a synchronization control channel by conveying interrupts in two directions through the FPGA logic and into dedicated off-chip wires.

In its first implementation this bridge on FPGA is clocked very conservatively at 20 MHz. This constitutes currently the main system bottleneck in our tests, non inherent to the STHORM architecture, which is intended for same-die integration with the host, with orders-of-magnitude faster bridge and larger memory bandwidth.

3. IMPROVING PROGRAMMABILITY OF THE STHORM SYSTEM

3.1 OpenMP Extensions

Traditionally, writing code for an heterogeneous SoC (e.g., with OpenCL) requires to manually write a program into separate files (at least one for the host, one for the accelerator), and to manually compile it into different binaries. The host program should also explicitly include instructions to load the accelerator binaries, to start the computation, to transfer data and to synchronize. In our proposal, the programmer writes a single OpenMP application, where a custom `offload` directive is used to abstract away all the procedure of i) outlining a program for the accelerator; ii) compiling it into a separate accelerator executable; iii) offloading code and data to the accelerator; iv) synchronizing.

```
#pragma omp offload [clause[[,]clause]...]
    structured-block
```

where *clause* is one of the following:

`name` (*string*`,`*integer-var*)
`private` (*list*)
`shared` (*list*)
`firstprivate` (*list*)
`lastprivate` (*list*)
`nowait`

The `name` clause is used to univocally identify a kernel to be offloaded to the accelerator. This is achieved through a literal (string) parameter, plus an integer variable whose declaration is visible from the code block immediately enclosing an `offload` directive. As we will explain later on, the string parameter is used during the generation of the accelerator program code and executable file. The integer variable is used for synchronization purposes. When the offload procedure is triggered, the accelerator is queried to verify its presence and availability to execute. If the request is successful, an integer value is returned, which specifies the unique ID of the offloaded job. This is necessary for two reasons. First, the `offload` directive can be coupled to a `nowait` clause, which specifies asynchronous operation mode (i.e., the CPU is not blocked waiting for the offloaded code to complete execution on the accelerator), thus allowing multiple offload requests to be enqueued on the accelerator. Second, the described offload technique supports multiple accelerators (likely to be common in future SoC evolutions), thus it is necessary that the returned ID encodes information about the physical device allocated for execution. The integer variable specified in the `name` clause is used to store the return value from an offload call. A negative return value indicates failure, thus the offload block is executed on the host itself. A positive return value indicates a valid offload context. The same integer variable specified in the `name` clause can be used to synchronize at specific program points with the custom `wait` directive

```
#pragma omp wait (integer-var)
```

Note that in case the `nowait` directive is not specified, the offloaded block executes synchronously (i.e., the CPU will block until completion of the accelerator execution).

The `private`, `shared`, `firstprivate` and `lastprivate` clauses can be used to specify data sharing between host and accelerator, and work pretty much in the same way of standard OpenMP. `private` variables are duplicated in the accelerator memory space (typically on a thread's stack, which is mapped on the L1 TCDM of the cluster hosting that thread). The code executing on the accelerator only refers to these private copies and does not access the host memory. `firstprivate` variables work in the same way, but they are initialized at the beginning of the offload block to the value of the original variables from the host execution context. Similarly, `lastprivate` variables have local storage in the accelerator memory space. Their content is determined during the execution of the offload block and copied back to the original variable in the host memory space at the end of the accelerator execution. `shared` variables identify truly shared main memory storage. Both the host and the accelerator directly access these locations.

Note that true variable sharing is possible in this implementation because the STHORM accelerator has physical connection (direct R/W access) to the main memory. This simplifies implementation of the relaxed consistency memory model. For improved performance, during offloaded code execution it is advisable to bring data onto L1 memories in the STHORM clusters via explicit DMA transfers. Similarly, if asynchronous offload is used and the host is allowed to touch the same shared data, it will bring it in L1 caches. In absence of hardware cache coherence, if this scenario is allowed by the programmer it becomes his/her own responsibility to ensure that appropriate data flushes to main memory are inserted both on the host and on the accelerator side whenever synchronization on shared data occurs.

Within an `offload` block all regular OpenMP constructs can be used. The STHORM accelerator is architected as a set of *clusters*, locally communicating via a fast crossbar-like interconnect (uniform latency) and globally via a NoC (non-uniform latency). This hierarchical interconnection system creates NUMA communication issues. *Nested parallelism* represents a powerful programming abstraction for cluster-based architectures, as it allows efficient exploitation of i) a large number of processors and ii) a NUMA memory hierarchy. Nested parallelism offers the ability of clustering threads hierarchically, which has historically played an important role for programming traditional cc-NUMA systems organized as clusters of multi-core computers. Here, outer levels of coarse-grained parallelism could be distributed among clusters, and data parallelism could be used to distribute work within a cluster.

Due to the relevance of affinity control in the context of ccNUMA machines, the OpenMP architecture review board has included in the recent draft specification v4.0 (public review release candidate) the definition of a new `proc_bind` construct, to be coupled to the `parallel` directive.

```
proc_bind ( master | close | spread )
```

The `master` policy assigns every thread in the team to the same place as the master thread. The `close` policy assigns the threads to places close to the place of the parent's thread. The master thread executes on the parent's place and the remaining threads in the team execute on places from the place list consecutive from the parent's position in the list, with wrap around with respect to the place list. The `spread` policy creates a sparse distribution for a team of T threads among the P places of the parent's place partition. It accomplishes this by first subdividing the parent partition into T subpartitions if T is less than or equal to P, or P subpartitions if T is greater than P. Then it assigns 1 (T≤P) or a set of threads (T>P) to each subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it is also a subset of places for a thread to use when creating a nested parallel region.

Figure 4 illustrates how `proc_bind` allows to easily map a nested parallel region over the target multi-cluster.

Our OpenMP compiler implements the proposed extensions, and relies on a custom runtime library for lightweight nested fork/join on the STHORM platform, based on our previous work [10] [11]. We allow to use all standard OpenMP constructs inside `offload` blocks, including the `proc_bind` directive to easily and conveniently achieve efficient execution of STHORM-accelerated code blocks.

3.2 Host Program Transformation

The transformation process described is based on extensions to the GCC (v4.5.3) OpenMP implementation [13]. Figure 5 shows a simple example program for the SoC host processor, which uses the described OpenMP extensions. The transformed code after OpenMP expansion is shown in

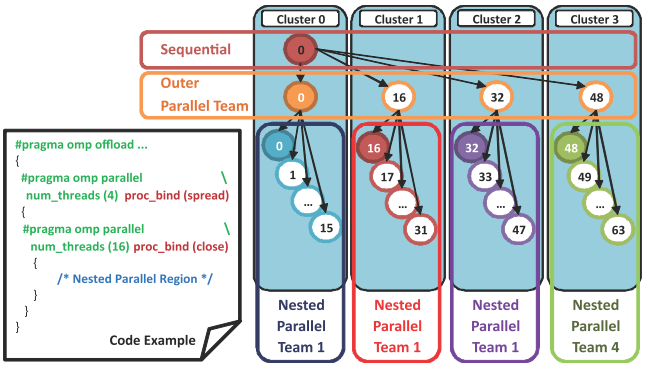

Figure 4: Nested parallel team deployment among clusters with the `proc_bind` clause.

```
void main(){
  int a[];
  int b[];
  int ker_id;

  /* some CPU code here */

  #pragma omp offload   \
        shared (a,b)    \
        name ("mykernel", ker_id)        \
        nowait
  {
     #pragma omp parallel sections       \
       num_threads(4)                    \
       proc_bind (spread)
     {
        #pragma omp section
        TASK_A();
        #pragma omp section
        TASK_B();
     }
  }

  /* some independent CPU code
     to run asynchronously here */

  /* sync with STHORM */
  #pragma omp wait (ker_id)

  /* more CPU code here */
}

TASK_A(){
  int i;
  #pragma omp  parallel for             \
      num_threads(16) private(i)        \
      proc_bind (close)
      for( i=0;…. )
         do_smthg(a[i], b[i], …);
}
```

Figure 5: A program with OpenMP extensions.

Figure 6. On the host side, the `offload` block is replaced by the code needed to handle the offload process, namely data marshalling to implement data sharing between the host and the accelerator and a call to the custom `GOMP_offload_task` runtime function. The `offload` block is outlined into a new function, similar to the expansion of `parallel` blocks, which is included into the call-graph compiled for the host target for conditional execution. If the value returned by the `GOMP_offload_task` is negative, the execution of the host version of the `offload` block is triggered.

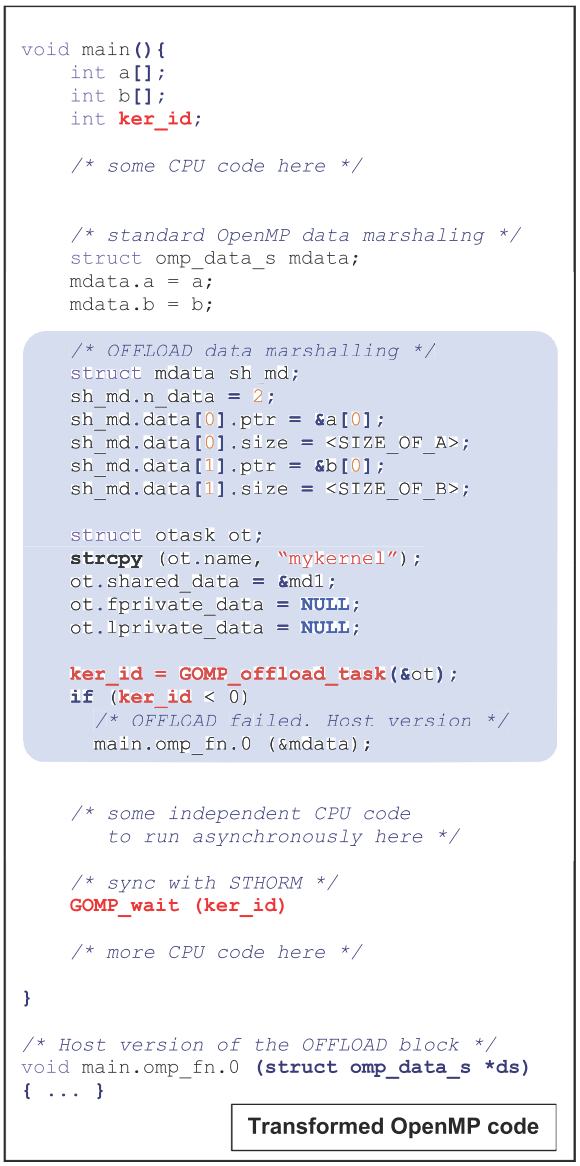

Figure 6: Transformed OpenMP program.

The marshalling procedure packs information about `shared`, `firstprivate` and `lastprivate` data into a data structure to be passed to the underlying software layer. Every data item is represented with a `data_desc struct`, which collects its base address pointer and size.

```
struct data_desc {
  unsigned int * ptr;
  unsigned int size;
}
```

A different array of `data_desc structs` is then collected for `shared`, `firstprivate` and `lastprivate` data, along with the number of instances of each type.

```
struct mdata {
  unsigned int n_data;
  struct data_desc data[n_data] ;
}
```

Finally, the `otask struct` gathers all these information, plus the kernel name.

```
struct otask {
   char *name;
   struct mdata *shared_data;
   struct mdata *fprivate_data;
   struct mdata *lprivate_data;
}
```

The compiler generates the code to populate this data structure from the calling context on the running host program, then passes it to the `GOMP_offload_task` runtime function.

This function is implemented as an extension of the OpenMP host runtime environment (RTE). Its simplified code is shown in Figure 7.

First, the name of the target kernel is extracted, and the corresponding object file name (`.so`) is resolved. Note that we also allow to produce the STHORM executable with just-in-time (JIT) compilation at this point. If this option is enabled, the name of the program in LLVM IR format is determined (`.llvm`), and passed to the LLVM-xp70 compiler.

After this step, the native STHORM runtime `CM_LoadInBanks` function is invoked, which dynamically links the kernel object and loads it into the accelerator L2 memory. Then, `firstprivate` data is handled. For each data element in the corresponding descriptor, enough memory is allocated from the STHORM heap to accommodate it, then a DMA transfer is triggered. The pointer to the STHORM copy is then inserted into a `context` data structure. For `shared` data no copy is involved, and only pointers to the host main memory are annotated into the `context` data structure. Finally, the `CM_CallMain` function is invoked to start the `main` method on STHORM.

The `main` method is implemented in the STHORM OpenMP runtime environment [10]. Here only one thread on the target cluster starts execution of the offloaded code region, until a `parallel` directive is encountered. This thread (the master thread) is pointed to `shared` and `firstprivate` data through the `context` data structure, while `lastprivate` data is by default mapped on its stack (or, explicitly, on the heap). In case of a synchronous offload, `lastprivate` data is copied back to the host main memory after the end of the STHORM kernel execution. When the `nowait` clause is specified, `lastprivate` data is dealt with inside the `GOMP_wait` primitive.

3.3 Multi-ISA Compilation Toolchain

Figure 8 depicts the multi-ISA toolchain that we realized for this work. The STxP70 toolkit is based on the Clang+LLVM [2] compilation infrastructure. However, neither Clang nor LLVM currently provide complete and robust support to OpenMP. On the contrary, GCC provides a mature and full-fledged implementation (GOMP) [13] of the OpenMP specification version 3.1.

DragonEgg [1] is a GCC plugin that replaces GCC's optimizers and code generators with those from the LLVM project. This provides a convenient starting point to our objectives. DragonEgg hooks to the GCC compilation pipeline after entering the AST-SSA form. All the optimization passes on the AST, and the target-specific backend compilation are thus done on LLVM.

Rather than disabling the GCC optimizers and backend, we create two distinct compilation flows. To do this, we modified DragonEgg as follows. First, the whole GCC compilation pipeline remains enabled to produce the final ARM host executable. Second, parts of the program, and specifically the `offload` blocks, are translated from the GCC GIMPLE-SSA IR into the LLVM IR and compiled into as many STxP70 executables.

To achieve this goal, we need to derive from the original

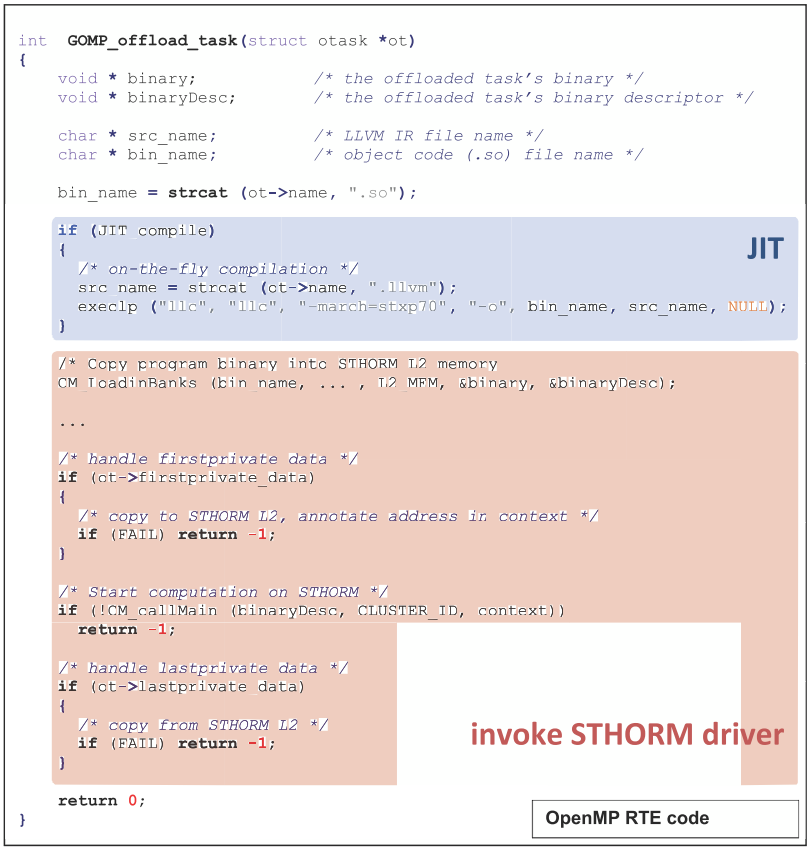

Figure 7: Runtime function for an offload.

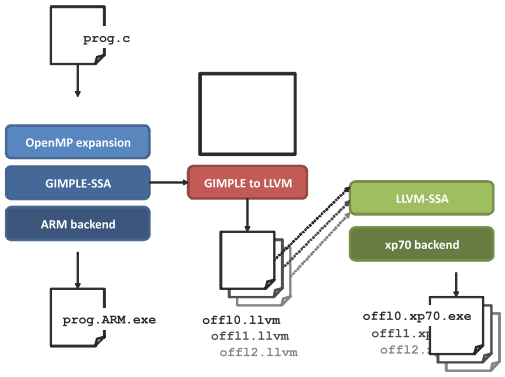

Figure 8: The multi-ISA toolchain.

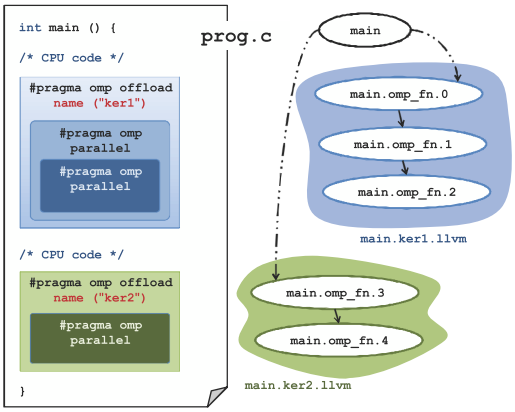

Figure 9: Deriving LLVM translation units for offloaded kernels from original call-graph.

program call graph as many LLVM translation units as the offload blocks. Figure 9 illustrates this process.

All the functions created in GCC by the expansion of OpenMP `offload` directives, as well as `parallel` directives and function calls therein, are marked with a `DECL_OFFLOAD` flag, which we use in our modified GIMPLE-to-LLVM translation pass in DragonEgg to filter out host program parts not meant to run on the accelerator. Every function tagged as `DECL_OFFLOAD` has also another attached *name* attribute: the string parameter specified with the `name` clause associated to the corresponding `offload` directive. Whenever a new function is processed, the value of the associated *name* is evaluated. The first time that a *name* is encountered, a new LLVM translation unit is created. All the functions with a same *name* attribute are appended to the sub-call-graph of the associated LLVM translation unit.

4. EXPERIMENTAL RESULTS

In this section we validate our programming model using three representative computer vision applications: color tracking, implemented from the open source computer vision library (OpenCV) for single color tracking, FAST [17] corner detection, and abandoned/removed object detection using a normalized cross-correlation (NCC) algorithm [9].

4.1 Parallelization Efficiency

As a first experiment, we explore the parallelization efficiency of OpenMP codes, locally on the accelerator (without the impact of the offload procedure). More specifically, we provide the speedup achieved by a parallel implementation of each of the benchmarks running on 64 cores, versus the same code running on a single STxP70. This includes the time for DMA transfers to/from the main L3 memory, but not the time spent on the host to initiate an offload sequence. To study how the interconnection towards the L3 memory through the FPGA bridge impacts the execution speed, we also run the same experiments on the STHORM heterogeneous system simulator (*GePop*), provided with the official SDK [12]. *GePop* allows to model the latency of the communication infrastructure from the STHORM chip to the main memory with a parameter, which was set to 100 cycles, as representative latency for a SoC where host and accelerator share external L3 memory though a multi-ported LP-DDR3 controller.

Figure 10 (left plot) shows the scalability of the parallelization for FAST, using a chess-pattern image of increasing size. We use this synthetic image for this experiment because it allows to keep the percentage of corners/pixel area constant as the image size is enlarged. This is an important thing to consider, because FAST has a data-dependent behaviour: a single iteration of the main kernel loop has a different duration depending on the fact that a corner is detected or not. The experiment with *GePop* indicates that for an image size close to the VGA resolution our implementation allows $> 53\times$ speedup versus single STxP70-execution. The same experiment with the board leads to a more modest $25\times$. This reduced speedup result is a consequence of the high latency, low bandwidth FPGA bridge, as confirmed by the speedup scaling plot on the right in Figure 10, where

we have repeated the previous experiment assuming that no communication to/from L3 is needed by the application (i.e., the data is assumed to be always already present in each cluster L1 memory). *GePop* faithfully models the behavior of the board in absence of L3 memory accesses.

Figure 11 shows the speedup achieved for the 64-core version of each of the benchmarks on *GePop* and on the board. Focusing on the *GePop* version, it is possible to see that our OpenMP implementation allows 30× speedup for Color Tracking and FAST, and ≈ 60× for NCC. Note that in this case we use a different image than the previous chess pattern for FAST (a real urban traffic scene, 640×480 format), with a much smaller percentage of corners. This creates more imbalance among the threads, due to the discussed data-dependent control flow, which justifies the smaller speedup achieved. The comparison with the board shows that the FPGA bridge induces up to ≈ 6× performance loss.

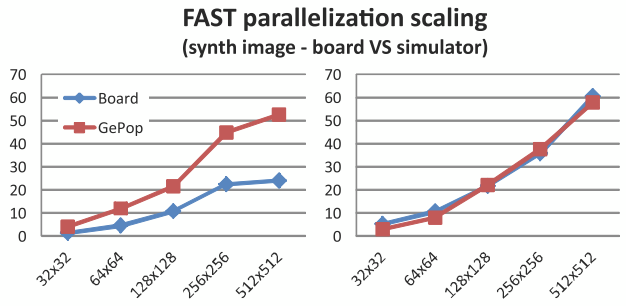

Figure 10: Left: 64-core to 1-core (xP70) speedup scaling for FAST, increasing image size. Board to simulator comparison. Right: Same speedup scaling experiment with no accesses to L3 memory.

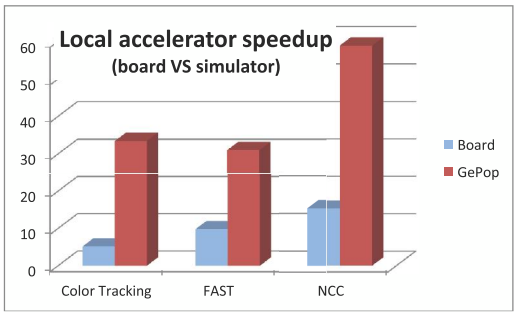

Figure 11: 64-core to 1-core (xP70) speedup for various benchmarks. Board to simulator comparison.

4.2 Comparison with OpenCL and Impact of the Offload

The second set of experiments compares the speedup achieved by our enhanced OpenMP implementation to hand-optimized OpenCL codes. The OpenMP and OpenCL accelerated versions of the benchmarks include the time for the offload procedure (copy of the STHORM executable on the accelerator, dynamic loading and initialization), measured from the host processor. Given the high impact of L3 memory cost in the board, and since the optimal data partitioning and double-buffering granularity schemes differ from the OpenCL to the OpenMP implementations – which may amplify the difference in performance achieved by the two – we measure the kernel time execution on *GePop*.

Figure 12 shows the results for this experiment, where the speedup numbers are normalized to the execution time of the sequential version of each benchmark running on a ARM core. On the X axis we consider an increasing number of processed frames, to have an estimation of the granularity of the offloaded job for which the cost of the offload itself is amortized. It is possible to see that for all the three benchmarks our OpenMP parallelization achieves nearly-identical performance to the hand optimized OpenCL codes. This result is obtained at a much lower programming effort thanks to the directive-based coding interface and to the efficient compiler and runtime implementation that we provided.

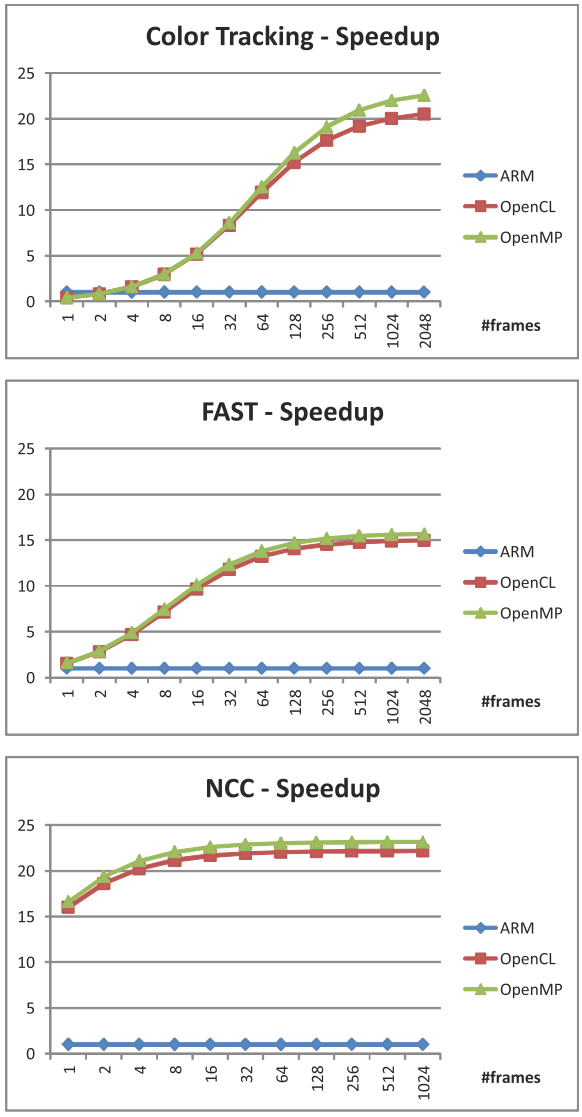

Figure 12: Speedup results.

5. RELATED WORK

Heterogeneous systems have been long since proposed as an effective design paradigm to improve the energy efficiency of embedded SoCs. Back in 2011 ARM launched its AMBA AXI4 on chip interconnection system [19], providing cache coherency not only among different cores of a main host multi-processor such as big.LITTLE [5] (which exploits itself a sort of heterogeneity in terms of performance and energy), but also among completely different IPs, such as DSPs or embedded GPUs (e.g., their MALI).

The relevance of heterogeneous IP integration for all ma-

jor SoC vendors is also witnessed by the Heterogeneous System Architecture foundation initiative (HSA)[20], a non-profit consortium of SoC IP vendors, OEMs, academia, OSVs and ISVs, whose goal is to drive all the aspects of heterogeneous computing, from programmability to architectural interface standardization.

There is a trend towards implementing more and more general-purpose (i.e., programmable) accelerators. General purpose GPU computing systems have witnessed this trend, but performance-wise they are not very efficient when dealing with multiple-instruction, multiple-data (MIMD) types of parallelism. This is pushing for the integration of programmable many-cores, both in the HPC domain (e.g., the Intel XEON-Phi [7]) and in the embedded domain (e.g., the STMicroelectronic STHORM [12]).

Clearly the increased system complexity has raised the need for more sophisticated programming models. OpenCL[8] was the first attempt to standardize programming for accelerator-based systems. OpenCL allows to achieve high performance, but requires significant effort. Its programming style is very low-level, and is not performance portable. Code optimizations remain specific to the considered acceleration device.

To simplify the programming interface, other approaches such as OpenACC [18] and PGI Accelerator [6] have borrowed the directive-based programming style of OpenMP. However, their interface is still sort of tailored to GPU-like accelerators, with a focus on loop-level parallelism.

Many-core accelerators do not have the same architectural limitations of GPGPUs related to MIMD parallelism, and thus the programming model should allow for easy coding of more irregular forms of parallelism to exploit them best. Our approach keeps the convenient directive-style proposed by these frameworks, but advances the supported parallel semantics of the target application by allowing the full OpenMP specification v3.1 to be executed on the STHORM accelerator. Affinity-control extensions achieve high performance on the NUMA-based architecture.

To the best of our knowledge, we are the first to propose an implementation of these extensions for a heterogeneous embedded system. Our work also introduces additional directives to control the accelerator from the main host program. In this respect, Cramer et al. analyze the cost of a similar extension proposed by Intel for their XEON-Phi processor [4]. From the point of view of the implementation, there are clearly significant differences with our approach. First, the Xeon-Phi coprocessor is based on the same ISA of the host system, thus the toolchain needs not to support multi-ISA compilation. Second, the OpenMP implementation running on the Intel coprocessor can leverage standard full-fledged operating system services, which are lacking on the STHORM platform, and which required ad-hoc design and optimization of the runtime system. Ayguadé [3] and White [21] also proposed OpenMP extensions to deal with heterogeneous systems. Their work is however mostly focused on syntax specification (and semantics definition), while implementation aspects and experiments are absent.

6. CONCLUSION

In this paper we have presented a programming model, compiler and runtime system for a heterogeneous system from STMicroelectronics featuring a ARM9 host processor coupled to a STHORM many-core accelerator. Our programming model is based on an extended version of OpenMP, where additional directives allow to efficiently offload computation to the accelerator from within a single OpenMP host program. A multi-ISA compilation toolchain hides to the programmer the cumbersome details of outlining an accelerator program, compiling and loading it to the STHORM platform and implementing data sharing between the host and the accelerator. Our experiments were conducted on a prototype board and on the heterogeneous system simulator provided within the official SDK. The results highlight that our implementation achieves very good parallel performance scaling, and that we are capable of achieving nearly-identical results to hand-optimized OpenCL codes, at a significantly lower programming complexity.

7. ACKNOWLEDGEMENT

This work was supported by EU projects VIRTICAL (FP7-288574) and MULTITHERMAN (ERC-291125).

8. REFERENCES

[1] DragonEgg - Using LLVM as a GCC backend. [Online]. Available: http://www.dragonegg.llvm.org/.

[2] The LLVM Compiler Infrastructure. [Online]. Available: http://www.llvm.org.

[3] E. Ayguadé et al. A proposal to extend the OpenMP tasking model for heterogeneous architectures. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] T. Cramer et al. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium*, 2012.

[5] P. Greenhalgh. ARM White Paper - Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7, 2011. [Online]. Available: http://www.arm.com/files/downloads/big.LITTLE_Final.pdf

[6] The Portland Group. PGI Compiler User's Guide. [Online].Available:http://www.pgroup.com/doc/pgiug.pdf.

[7] A. Heinecke et al. From GPGPU to many-core: Nvidia Fermi and Intel many integrated core architecture. *Computing in Science Engineering*, 14(2):78–83, 2012.

[8] Khronos OpenCL. http://www.khronos.org/opencl/.

[9] M. Magno et al. Multi-modal Video Surveillance Aided by Pyroelectric Infrared Sensors. In *Workshop on Multi-camera and Multi-modal Sensor Fusion Algorithms and Applications - M2SFA2 2008*, 2008.

[10] A. Marongiu et al. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 105–110, 2012.

[11] A. Marongiu et al. Supporting OpenMP on a multi-cluster embedded MPSoC. In *Microprocessors and Microsystems*, 35(8):668 – 682, 2011.

[12] D. Melpignano et al. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM.

[13] D. Novillo. OpenMP and automatic parallelization in GCC. In *GCC developers summit*, 2006.

[14] Open Multi Processing. http://www.openmp.org.

[15] A. Rahimi et al. A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[16] J. Reinders. Intel White Paper - An Overview of Programming for Intel Xeon processors and Intel XeonPhi coprocessors. [Online]. Available: http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors.pdf, 2012.

[17] E. Rosten et al. Faster and better: a machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32:105–19, 2010.

[18] The OpenACC standard. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[19] A. Stevens. ARM White Paper - Introduction to AMBA 4 ACE. [Online]. Available: http://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf, June 2011.

[20] The Heterogeneous System Architecture Foundation. [Online]. Available: http://www.hsafoundation.com.

[21] L. White. OpenMP extensions for heterogeneous architectures. In *Proceedings of the 7th international conference on OpenMP in the Petascale era*, IWOMP'11, pages 94–107, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] www.OpenMP.org. OpenMP Application Program Interface v.4.0. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf.